

Advanced Audio Genomics Program Build Plan

Dynamic Tone Generation (DNA/RNA Frequency Synthesis)

Each DNA/RNA base will be synthesized as a precise audio tone according to the waveform assignments and frequencies from the provided reference data. The program must generate these tones with extremely high precision (up to 38 decimal places for frequency) to capture the subtle microtonal differences in the DNA base spectra . The tone generator will output raw audio waveforms for each base in sequence, inserting specified separator tones between them. Key implementation steps include:

- **Base-to-Waveform Mapping:** Map each nucleotide base to its waveform type: Adenine = sine wave, Cytosine = sawtooth, Guanine = triangle, Thymine = square, Uracil = impulse. Treat Uracil as equivalent to Thymine for mapping purposes (use the same square/impulse waveform logic for both, since U replaces T in RNA). For example, if processing an RNA sequence and encountering U, generate the same waveform type as for T (a square wave or short impulse pulse).
- **Frequency Data Integration:** Load the specific frequency values for each base from the provided DNA/RNA frequency data (over 60 distinct frequencies measured in the reference PDF). These frequencies correspond to infrared absorption spectra scaled into audible range . Store them in a lookup table (e.g. a dictionary mapping base+mode index

to frequency). Each base has multiple resonance frequencies; use the full set for richness or a primary frequency as needed. Ensure the frequencies are represented with high precision (e.g. using Python's Decimal or double-double precision floats) so that generated tones match the exact values (e.g. Adenine ~545.6000000000000... Hz, etc.) without rounding error.

- **Tone Synthesis Engine:** Implement a tone generator that can produce waveform samples at a given frequency and duration. Use an efficient audio synthesis library or algorithm: for instance, SoX's synthesis functionality or a DSP library to generate waveforms at runtime. Each waveform type (sine, square, triangle, sawtooth, impulse) can be generated via math formulas or signal libraries (e.g. use a sine function for Adenine, a piecewise linear function for triangle, etc.). If using an external tool like SoX, the program can invoke it to synthesize tones (e.g., `sox -n -r 192000 -b 32 tone.wav synth <dur> sine <freq>` for a high-res sine) . Otherwise, generate the wave by iterating at the target sample rate (e.g. 192 kHz) and computing each sample.
- **Cycle Precision and Duration:** For each base, play the tone for exactly 3 cycles of its frequency. Calculate the period $T = 1/\text{frequency}$ and set the tone duration = $3 * T$. This means higher-frequency bases have shorter duration tones than lower-frequency ones, preserving the cycle count uniformity. Because each base's tone may only last a few milliseconds (depending on freq), maintain sample-accurate timing so that 3 full oscillation cycles are generated.
- **528 Hz Impulse Separators:** Insert a brief 528 Hz impulse wave between bases and triplets to act as audio punctuation. Specifically, after generating the 3-cycle tone for a base, generate a **528 Hz** waveform of 1

cycle duration (≈ 1.89 ms) as a separator before the next base. After each codon (every group of three bases), insert a longer 528 Hz break of 3 cycles (≈ 5.68 ms). Use an **impulse waveform** for these separators: e.g. a single-cycle sine or a short square pulse at 528 Hz to create a click/tick sound. This will create a rhythmic structure where bases and codons are clearly delineated by the 528 Hz clicks. Ensure these impulses are also synthesized at high precision (528.000... Hz exact). The 528 Hz tone is significant (often called the “DNA repair” or Solfeggio frequency) and serves as a carrier binding tone in later modulation stages.

- **Memory and Performance:** To handle potentially millions of bases efficiently, build the tone sequence in a streamed manner or using buffers. For example, process the input sequence in chunks rather than constructing one giant waveform in memory. Use NumPy arrays or an audio buffer to accumulate samples. This allows the program to scale to large genomes or big files without running out of memory. The tone generator should be optimized with vectorized operations or use of a DSP library so that even at 192 kHz sample rate and 32-bit depth the synthesis is done quickly and precisely.

Binary to DNA Conversion & Hebrew Encoding

The program will accept any file type as input and convert its data into a DNA/RNA base sequence before tone generation. Two conversion pathways are provided: a direct binary-to-base mapping for non-text data, and a special text encoding that involves Hebrew letter mapping via Gematria. This ensures both arbitrary binary data and meaningful text can be translated into genetic sequences. The conversion logic works as

follows:

- **Binary-to-Base Mapping:** For arbitrary binary files (non-text data like images, executables, etc.), perform a straightforward bitwise mapping to the four nucleotide bases. Use 2 bits per base (since $2^2=4$ bases). Define the mapping as: 00 -> A (Adenine), 01 -> C (Cytosine), 10 -> G (Guanine), 11 -> T (Thymine) (for DNA mode) or 11 -> U (Uracil) in RNA mode. The converter should read the file's bytes, extract each bit pair, and map them to the corresponding base. For example, a byte 0xE3 (binary 11100011) would be split into bit-pairs: 11 10 00 11 which maps to bases: T, G, A, T. Assemble all bytes into a continuous base string. **Note:** If the input is explicitly to be treated as RNA output, use U in place of T for any 11 pair.
- **Text-to-Base Conversion:** If the input file is detected as text (e.g. a .txt file or UTF-8 text content), use the **Hebrew encoding via Gematria** as an intermediate step instead of raw binary mapping. The goal is to represent linguistic content in DNA form by encoding each textual letter or character as a group of three bases (a codon) corresponding to a Hebrew letter. The process is:
 1. **Language Translation (if needed):** First, interpret the text content. For non-Hebrew input, translate the text into Hebrew using an offline algorithmic translator. Integrate a free offline translation library such as **Argos Translate**, which supports many languages (Arabic, Chinese, English, French, Hebrew, etc.) . This allows global text input to be converted into Hebrew script internally. For example, English input “life” would be translated to the Hebrew word for life (“חיים”), preserving meaning. (If the text is already in Hebrew or the user chooses to skip translation, you can use it directly.) This

translation step ensures the meaning is preserved and prepares the text for Gematria mapping.

2. **Hebrew Letter to Codon Mapping:** Choose a scheme to map each Hebrew letter to a unique DNA triplet (3-base sequence). There are 22 letters in the Hebrew alphabet (plus 5 final forms), which can be accommodated within the 64 possible codons (4^3). One approach is to assign codons based on traditional Hebrew **Gematria** values or alphabet order. For example, Alef (א) could map to AAA, Bet (ב) to AAC, Gimel (ג) to AAG, and so on through the alphabet. Alternatively, use the Gematria numeric value of the letter (Alef=1, Bet=2, ..., Tav=400) mod 64 to pick a codon. **DNA mode (standard Hebrew):** Use a normal mapping (e.g. Alef through Tav in order). **RNA mode (inverted Hebrew):** Use an inverted mapping (e.g. Alef mapped to what Tav is in DNA mapping, Bet to Shin's codon, etc.), effectively reversing the alphabet or using the Atbash cipher convention. This means if DNA encoding yields a certain Hebrew letter for a codon, the RNA encoding yields the "opposite" letter for the same codon. By doing so, a DNA-coded message and its RNA-coded counterpart are related by an inverted Hebrew substitution.
3. **Text Encoding:** Convert each translated Hebrew character into its 3-base codon using the mapping from step 2. If the text has spaces or punctuation, you might define special codons or ignore them (or separate words by a specific codon sequence). The output will be a DNA base string that encodes the entire text message in a series of codons. For example, the Hebrew word מים might be encoded as ACG-TTA-GCC-... (each 3-base group corresponding to מ , ' , נ

respectively) according to the chosen map.

- **Fallback for Non-ASCII Data:** If the text is in a complex script or contains characters not handled by the translator (e.g. emoji, etc.), default to binary encoding for those parts or strip unsupported symbols. The primary focus is on alphabetic content which can be translated.
- **No Hebrew for Binary Files:** For non-text files (or if the user chooses not to do linguistic encoding), **skip the Hebrew step**. In those cases, do not translate or map to Hebrew letters at all; simply apply the direct binary-to-base conversion described earlier. This ensures that purely binary data (e.g. an image) is encoded as a valid base sequence without trying to force it into a language model.
- **Output DNA/RNA Sequence Preparation:** Whichever path is taken, the result is a linear sequence of bases (A,C,G,T or U). This sequence will then be fed into the tone generator. Before synthesis, allow the user to choose whether the output should use T (DNA alphabet) or U (RNA alphabet) for the thymine/uracil representation. The conversion logic can easily swap all T to U if RNA output is desired. The Hebrew inversion mapping should correspond accordingly (i.e., if RNA mode, invert the letter mapping and use U in the sequence).
- **Verification:** Implement checksums or simple verifications on the conversion (like showing a snippet of the generated base sequence to the user, or even reversing the process for verification in debug mode). This ensures that text conversion to Hebrew codons or binary mapping has occurred correctly. For instance, if a text is encoded to DNA and back-translated, it should recover the original text (for a sanity test of the mapping tables).

Multi-Stage Audio Modulation (FM & AM Integration)

After generating the base frequency audio (the “audio genomics” signal), the program will embed this signal into a musical carrier through two stages of modulation: first **FM (Frequency Modulation)** using a 528 Hz carrier wave, then **AM (Amplitude Modulation)** using an external music file as the carrier. This dual modulation process hides the genomic frequencies within a musical track in a structurally layered way. The implementation is as follows:

- **528 Hz Carrier FM Modulation:** Use the 528 Hz tone as a constant carrier and **FM-modulate** it with the DNA/RNA audio signal. In practice, this means the instantaneous frequency of a 528 Hz sine wave will be altered slightly according to the amplitude of the genomics signal. The DNA audio (which contains frequencies largely in the hundreds of Hz range) acts as the modulating signal. To implement FM: generate a high-sample-rate sine wave at 528 Hz and continuously adjust its phase/frequency based on the genomics signal. Use a modulation index that yields an audible but not overly drastic frequency deviation (this can be tuned experimentally – e.g., a small frequency deviation like ± 5 Hz or ± 10 Hz around 528 for subtle embedding, or more for pronounced effect). Libraries like **Liquid-DSP** (C library) can handle analog FM modulation easily, or you can implement it manually by phase accumulation:
$$\text{phase_increment} = 2\pi * 528\text{Hz} / F_s + k * (\text{mod_signal_sample}),$$
where k scales how strongly the genomics audio shifts the carrier frequency. The output of this step is an FM-modulated waveform centered at 528 Hz that carries the DNA pattern in its frequency variations.

- **Music File Ingestion:** Allow the user to provide a music file (any common format like WAV, MP3, etc.) to serve as the final carrier. Load this music track using an audio library (e.g. **pydub** or **Librosa**). Ideally convert it to a high-fidelity format internally (e.g. 32-bit float PCM) to mix with the generated signal without quality loss. If the music is stereo, decide whether to embed the genomic signal in one or both channels (embedding in both channels or just one channel could be a user option).
- **Automatic Pitch Analysis:** Analyze the music track to detect its tuning reference and dominant pitches. Using **Librosa** or a similar music analysis tool, estimate the tuning deviation from A440. Librosa's `piptrack` or `estimate_tuning` functions can find if the music is tuned to A4=440 Hz or slightly off. If it's not already at 432 Hz, compute the adjustment needed. For example, if the track is standard 440 Hz tuning, the adjustment factor to 432 Hz is ~ 0.9819 (since $432/440$). Detect the key or scale of the music as well by analyzing the pitch class distribution or using music theory libraries – this can inform how to blend the DNA frequencies harmonically.
- **Optional Retuning to 432 Hz:** If the user opts for 432 Hz tuning (as specified), apply a pitch shift to the entire music track to retune A4 to 432 Hz. This can be done by resampling the audio to a slightly lower sample rate or using a high-quality pitch-shifting algorithm (Librosa can resample with minimal quality loss). For instance, if a song is in A=440, lower the pitch by ~ 32 cents (hundredths of a semitone) to reach 432 Hz reference. This ensures the carrier music is in the desired tuning which is often considered more “harmonic.” Confirm the retuning by checking that an A in the music now corresponds to 432 Hz frequency (Librosa provides conversion utilities for tuning offsets).

- **Frequency Matching and Locking:** Adjust the **genomic frequencies** to better mesh with the music. There are two main strategies: (a) **Key alignment:** shift the set of DNA base frequencies collectively so that their central frequency (e.g. that C# ~544 Hz discovered in DNA) aligns with a musically relevant frequency in the track (perhaps the tonic or dominant of the song). For example, if the music is in the key of C (261.6 Hz base), you might transpose the genomic frequencies down so that 544 Hz (C#5 in 440-tuning) becomes 523.3 Hz (C5 in 432-tuning) or an appropriate note in the scale. (b) **Note quantization:** map each DNA frequency to the nearest note in an equal-tempered 432 Hz scale or to the nearest harmonic of the music playing at that moment. For instance, if a particular DNA tone is 546 Hz but the song chord contains a 544 Hz pitch (C#5 in 432 tuning), you might snap the DNA tone to 544 Hz for consonance. This **auto note-matching** can be done in real-time: use a short-time Fourier transform on the music to identify prominent frequencies, and when generating the DNA tone that overlaps that time, adjust it to the closest prominent frequency. This step is complex, but even a simpler approach like shifting all DNA tones by a constant factor to match the music's tuning can greatly improve harmony.
- **528 Hz FM Signal Integration:** Take the FM-modulated 528 Hz signal (from the first step) and prepare to embed it into the music. This signal by itself may not be audible if the frequency deviation is small around 528 Hz (which is in the upper mid-range of hearing). The next step is to use the music as a carrier via amplitude modulation.
- **Amplitude Modulation onto Music:** Perform **AM modulation** where the music track is the carrier and the FM DNA signal is the modulator (informational signal). In practice, multiply the music waveform by a

subtle varying gain derived from the FM signal. First, normalize the FM signal to have values roughly between -1 and +1 (or 0 to 1 if unipolar). Then decide on a modulation depth – since we want the genomic audio to be *subaudible*, the depth will be small (e.g. 5% variation). For example, use a depth range such that the music's amplitude oscillates ± 1 dB or so according to the DNA signal. Compute the AM waveform: $\text{output} = \text{music} * (1 + \text{mod_depth} * \text{fm_signal})$. If the FM signal is bipolar (-1 to 1), you may use $(1 + m * \text{depth})$ where $\text{depth} \sim 0.05$, ensuring it never goes negative (clamp at 0). This will embed the DNA frequencies into slight volume fluctuations of the music. Because the DNA signal itself has a 528 Hz carrier in it, what actually happens is that the music's amplitude will subtly pulsate at rates determined by the DNA signal's frequency pattern (essentially imprinting the pattern onto the music's amplitude). This is analogous to how radio amplitude modulation works, but here the effect is kept very low in magnitude to remain subliminal.

- **Subaudible Level Adjustment:** It's crucial to set the modulation depth such that the genomic signal is **-33 dB to -55 dB** relative to the main music. This range is specified to keep the embedded data below the threshold of conscious hearing. After modulation, analyze the mixed signal's levels: ensure that the variations due to DNA do not cause audible artifacts. You can measure the resulting signal's spectral profile to confirm that no obvious new tones are sticking out above the noise floor of the music in that -33 to -55 dB range. If they are, reduce the depth. Essentially, the DNA patterns become a form of **audio watermark** in the music. The user won't clearly hear them, but theoretically the information is present and could be revealed by filtering or further analysis.
- **Stereo and Phase Considerations:** If the music is stereo, you may

choose to embed the DNA signal differently in left vs. right channels (e.g. invert phase on one channel's modulator for a subtle stereo effect, or embed in just one channel). This can make the embedded signal even less noticeable, as the human ear might cancel out perfectly out-of-phase low-level signals. However, ensure this doesn't interfere with the desired effect (maybe allow the user to select stereo embedding or mono).

- **Implementation Libraries:** Use **pydub** or **AudioSegment** to perform the amplitude modulation and mixing, as it simplifies handling multi-format audio. You can get raw audio arrays via pydub for the music and then multiply by the modulator values (which you generate as an array from the FM step). Librosa or NumPy can also be used for sample-level operations with ease. If liquid-dsp (C library) was used for FM, its output can be brought into Python as a NumPy array of samples to continue the AM step. The entire modulation chain can thus be: base sequence -> tone generation (NumPy array) -> FM mod (NumPy or liquid-dsp) -> modulation array -> multiply with music array -> output.
- **Testing the Modulation:** It's a good practice to test this pipeline with a simple music input (like a pure tone or a simple melody) and a short DNA sequence to verify that the FM and AM are working. For example, use a simple music sine wave at 440 Hz and embed a known DNA pattern – then use a spectrum analyzer to see that sidebands or amplitude variations at expected frequencies appear around the music frequency, but at very low magnitude. This will confirm the genomic signal is encoded in the output.
- **Result:** The final output of this stage is a new audio signal (stereo or mono) that sounds like the original music (possibly retuned to 432 Hz) with no obvious additions, yet contains the DNA frequency data hidden in

its amplitude fluctuations and centered on a 528 Hz FM carrier. Conceptually, it's similar to an FM radio signal being amplitude-modulated onto a song – a two-layer carrier that binds the DNA “code” into the music .

Adaptive Scaling and Multi-Layer Composition

To accommodate input files of vastly differing lengths and to mirror biological genome organization, the program includes logic for scaling the audio genomic sequence to the music length and for combining multiple sequences into layered structures (like chromosomes). This ensures that the audio output can handle small or extremely large data gracefully, either by looping or speeding up the genomic pattern, and by splitting data into parallel layers if needed. The strategy is as follows:

- **Duration Matching (Looping):** After generating the initial DNA audio sequence, compare its duration to the music track's duration. If the genomic audio is shorter than the music, implement looping. Continuously repeat the base frequency pattern (including the 528 Hz separators) to fill the entire length of the music track. For example, if the DNA sequence produces 30 seconds of audio and the music is 3 minutes, loop the sequence 6 times back-to-back. Ensure there is a seamless transition between loop iterations (the end of one sequence should flow into the beginning of the next without a click; you can omit the final separator at the very end of each loop iteration to avoid doubling it). This looping guarantees that the modulation can persist throughout the whole song. Optionally, you might introduce slight phase shifts or crossfades between loops to avoid a sharp repetitive boundary.

- **Speed-Up for Long Sequences:** If the genomic audio is longer than the music track (e.g., encoding a large file could produce minutes or hours of audio, but the chosen song is shorter), apply **exponential time compression** to fit it in. The guideline is to use base-8 scaling factors: speed up by $8\times$, $64\times$, $512\times$, etc., until the DNA audio length is equal to or just shorter than the music length. For instance, if the DNA audio is 10 minutes and music is 2.5 minutes, a $4\times$ speed-up isn't enough (would give 2.5 min), but $8\times$ yields 1.25 min which fits (though shorter; one could loop it then). Use 8^n factors to preserve some harmonic relationships in the time domain (powers of 2 might preserve octaves in frequency). Implement this by increasing the playback sample rate of the DNA audio or by algorithmically skipping samples. Libraries like **pydub/AudioSegment** or **Librosa** can change playback speed by resampling. Be mindful that extreme speed-ups ($64\times$, $512\times$) will raise the frequency of every tone by the same factor, potentially moving some frequencies out of the audible range. However, moderate factors will likely keep them mostly audible. The idea is that the pattern of frequency changes (the relative melody of the DNA) is preserved, just accelerated. Inform the user when such scaling is applied (perhaps in the UI, show "Data length exceeded music, applied 8x compression").
- **Hierarchical Multi-Layer Modulation:** For very large data sets (for example, encoding entire chromosomes or multiple files), the program should utilize **multi-layer modulation** instead of trying to force everything sequentially. Inspired by how real chromosomes are organized (with multiple chromosomes in a genome, each containing many genes), we create multiple layers of audio, each carrying part of the data, all combined in the final output.

- **Data Splitting:** If a single file's data is extraordinarily long, split the base sequence into several chunks (e.g., split into N segments of roughly equal length). Each chunk will be treated as a separate "layer." Similarly, if multiple files are provided (batch mode or a directory of files), each file's sequence can naturally be one layer. Label these layers as if they were different chromosomes (Chromosome 1, 2, 3, etc. or Layer A, B, C...).
- **Parallel Layer Generation:** Generate the frequency audio for each layer separately (applying the same base tone rules). Now you have, for example, Layer1 audio and Layer2 audio (and more, as needed). Each layer is typically as long as the full input data requires; if layers differ in length, you can loop or truncate them to a common length for combination or treat the longest as the reference.
- **Harmonic Layer Alignment:** If multiple layers play simultaneously, it's important to ensure they do not create cacophony. Consider allocating each layer a different **frequency "channel" or carrier offset**. For example, use different base carriers for FM modulation: one layer could use 528 Hz FM, another could use a higher carrier like 639 Hz (another Solfeggio frequency), etc., so that their information is somewhat separated in frequency domain. Alternatively, stagger them in time or stereo field. However, the spec suggests using **AM for structuring layers** and **FM for 528 Hz binding**. This could mean that layers are combined via amplitude modulation in a nested fashion:
 - For two layers: take Layer1's FM-modulated 528 Hz signal and amplitude-modulate it by Layer2's FM signal (or vice versa). In

other words, use one layer's modulated output as the carrier and impose the second layer onto it as an amplitude variation. This creates a single composite signal where one layer might modulate the other's amplitude envelope. If more layers exist, this process can be iterative (Layer1 FM output AM-modulated by Layer2, then that result AM-modulated by Layer3, etc.), building a complex modulation hierarchy. Each additional layer adds another amplitude modulation level, structuring the data in nested form. This is analogous to how chromosomes (layers) could be intertwined – one within another – in the final audio.

- Another approach: simply mix the layers additively but at different subaudible volumes (e.g., Layer1 at -40 dB, Layer2 at -45 dB, etc.) and possibly panned differently. This is simpler but may cause frequency interference if two layers have similar frequencies. The **AM structuring** method avoids direct frequency interference by using amplitude domain to combine.
- **Implementation:** If using nested AM, generate each layer's FM signal (centered on 528 Hz or different carriers if chosen). Then perform amplitude modulation layering: treat Layer1 FM output as a base, multiply it by $[1 + \text{depth2} * (\text{Layer2 FM signal})]$ to embed layer 2 into layer 1. If more layers, continue: the result now carries layers 1 and 2, use it as base and modulate with layer 3 signal, and so on. Keep depths small to maintain subaudible levels for higher layers. This yields one complex signal carrying all layers. If using simple mixing, just ensure each layer is scaled to a low volume and sum them; perhaps use slight EQ filtering to give each layer its own “band” (e.g., layer1 frequencies might span one range, layer2 another, if their

frequency content differs). In either case, all layers should be ultimately bound by the 528 Hz FM carrier (since each layer's data was FM'd onto 528 or a related frequency, and combining them still results in a composite around that area). This satisfies the idea of "528 Hz binding" – 528 Hz remains a central reference tone threading through the whole modulated structure.

- **Chromosome-like Output:** If the user provided multiple files or a structured directory, consider outputting each layer as a separate track or combine into one track with multi-layer modulation as above based on user preference. The UI could allow toggling layers or soloing a particular layer's contribution. But by default, for a single music output, we embed all layers into that one track via modulation as described.
- **Biological Compression Logic:** Apply logic to compress repetitive data segments in a biologically inspired way. DNA often has long repeats which nature "compresses" by structural coiling, etc. In audio, if the input data has a very long repeated pattern, we can choose to not literally repeat the entire pattern in audio. Instead, generate one instance of the repetitive sequence's audio and then **symbolically loop it via modulation**. For example, if a sequence of bases repeats 1000 times, you might generate the audio for that sequence once, and then rather than playing it 1000 times, you insert a special modulation marker that indicates repetition (perhaps a unique tone, like the 555 Hz "N" tone, or a slight pause, to signify a large skip). However, this is an advanced feature and might be optional. At minimum, document that the system can detect large repetitive elements (like large sections of identical bytes) and handle them efficiently (maybe by speeding them up more, or by layering

a shorter representative signal).

- **Example of Multi-layer:** Suppose the user has 3 text files (e.g., chapters of a book). The program translates each to DNA sequences (layer1, layer2, layer3). It then chooses to treat these as three chromosomes. To embed in one song, it FM modulates 528 Hz with each layer separately. Then perhaps sets layer1 to modulate the left channel, layer2 the right channel, and layer3 through nested AM on both, or some combination such that all three are present but not interfering. Alternatively, it could produce three separate audio outputs (one per layer) if the user wanted each chromosome in a different music piece. But the instructions lean toward combining them (“harmonically layered”), so likely the single-song approach with layering is intended.
- **Synchronization:** When layering, ensure that the start of each layer’s sequence aligns in a meaningful way. You could start them all at time 0 together (parallel embedding), or maybe stagger their start times to reduce instantaneous load (like layer2 starts when layer1 is halfway, etc.). For nested AM, they inherently start together in the modulation. It might be wise to align them so that codon boundaries across layers occur simultaneously if possible (for example, all layers insert their 528 Hz separators in phase), to create a coherent multi-layer “beat.” This would mirror how chromosomes might line up on certain structural cues.
- **Performance Consideration:** Combining layers multiplies the data to process. Use multithreading or multiprocessing to generate each layer’s audio in parallel, leveraging multiple CPU cores (e.g., one thread per layer computing the tone sequence). Once generated, the combination (AM or mixing) is relatively quick. Also, provide an option in the UI to limit the number of layers if a machine can’t handle too many heavy

operations at once.

- **Output Confirmation:** The final composite modulated signal (after layering and modulation) should be checked to ensure it still lies in the desired subaudible embedding range. Because adding signals can raise overall volume, you might need to normalize the final output to avoid clipping, and then adjust the embedded signal's level back to the target range. Always preview the result (perhaps a quick Fourier analysis) to ensure layers didn't produce unexpected audible artifacts.

Gene Sequence Analysis and Synthesis Tools

This module provides bioinformatics capabilities: searching for real genetic sequences online, converting between protein and DNA representations, and allowing direct user editing of gene sequences. It enriches the program by linking it with actual genomic data sources and giving the user fine control over sequences.

- **PubMed/GenBank Integration:** Incorporate a feature to fetch genetic information from online databases (NCBI's GenBank for sequences, PubMed for literature) via their APIs. Using Biopython's Bio.Entrez module is an effective approach. This allows programmatic search and retrieval of nucleotide or protein sequences by accession number or keyword . For example, provide a search bar where a user can input a gene name (like "BRCA1 human") or a GenBank ID (like "NM_007294"), and the program will query NCBI and retrieve the corresponding DNA sequence data. The Entrez EFetch API can return the sequence in FASTA or GenBank format , which the program can parse (Biopython can

parse the result into a sequence object). Similarly, a PubMed search could retrieve abstracts for a given gene or topic (though that's more for informational display than audio). This feature effectively lets the user *sonify* real genes: the program can take the fetched DNA sequence and immediately convert it into the base frequency audio (using the core features above). Include necessary input fields for API keys or email (NCBI requires an email parameter and has request limits).

- **Protein-to-DNA Conversion:** Enable input of protein sequences (amino acid sequences) and convert them back to DNA or RNA coding sequences. A protein is typically given as a string of one-letter amino acid codes (20 standard amino acids). To convert to DNA, use the standard genetic code to map each amino acid to a codon. This mapping isn't one-to-one (most amino acids have multiple possible codons), so the program should use a default codon table (e.g., the standard codon table where, say, Phenylalanine (F) maps to TTT or TTC – choose one consistently, perhaps the first alphabetically or biologically common codon for each). For example, if the user inputs a protein sequence “MAG”, the program could output a DNA sequence “ATGGCGGGG” (ATG -> M, GCG -> A, GGG -> G). Provide an option to choose DNA or RNA output (i.e., use T or U). If any ambiguous or stop codon appears in the protein sequence input, handle it: e.g., * for stop could map to a stop codon like TGA, or just be skipped.
- **Support for 'N' Base:** In some cases, sequences (from GenBank or user input) may contain the nucleotide N which means “any base”. This program defines a special audio representation for N: a 555 Hz tone (the spec calls for N = 555 Hz). Implement this by treating N as its own base type. In the tone generator, map N to an impulse or sine at 555 Hz

(perhaps use an impulse wave similar to how Uracil was considered an impulse). This tone will signify an unknown or wildcard base. If converting protein to DNA and encountering an ambiguity (like an unknown amino acid or one that could be coded by multiple codons), you might insert an N in the DNA sequence to denote uncertainty, which then produces the 555 Hz sound. Also use the N tone if any web-retrieved sequence has ambiguous letters (sometimes sequences have other ambiguity codes like R, Y, etc. – you can map all non-ATCG explicitly to some pattern or default to N = 555 Hz to handle them uniformly).

- **Manual Gene Editor:** Provide a GUI text editor component where users can manually input or tweak a nucleotide sequence. For example, after fetching a gene from GenBank, display the sequence in a text area (possibly formatted in codon groups or with an index). The user can then edit bases (e.g., mutate a base, or paste in a new sequence). This manual edit will override or create a new sequence for sonification. Features for this editor might include: validating that only A, C, G, T (or U, N) characters are entered (alert or color the text if invalid chars present), an option to insert a specific codon or base tone, and perhaps a small library of example sequences. Once the user is done editing, they can click “Generate Audio” to use the edited sequence for tone generation. If the sequence length is not a multiple of 3 and the Hebrew text mapping is in use, handle that (the Hebrew mapping expects triplets – but since manual editing likely is for DNA directly, not relevant to the text mapping scenario).
- **Visualization & Confirmation:** It might be useful to show a simple visualization of the sequence’s audio profile – for instance, a small plot of the frequency pattern or a list of the base->frequency mapping for the

sequence. This can be part of the gene analysis module: after conversion, list each codon and its corresponding Hebrew letter (if text mode) or each amino acid and chosen codon (if protein mode), etc., so the user has a record of what is being sonified. Additionally, if a sequence was fetched from GenBank, show its description (from the GenBank record) so the user confirms it's the right gene.

- **Integration with Audio Generation:** The gene analysis tool ties into the main program flow by providing another way to get sequences. Once a sequence is fetched or edited, the user should be able to send it to the main pipeline (perhaps by clicking “Add to project” or “Convert to Audio”). That sequence is then treated like any other input: it can be assigned to a layer or output to a music track. For example, a user might fetch two different gene sequences and choose to layer them together in one song.
- **Efficiency & API Usage:** When using the NCBI API, respect rate limits (use an email and API key, and don't flood requests). Since this is an offline application scenario, you might prompt the user to ensure internet connectivity for this feature. If working offline, perhaps allow the user to load a local FASTA file as an alternative.
- **Example Use Case:** User searches “BRCA1 gene”. Program uses Entrez ESearch/EFetch to retrieve the BRCA1 DNA sequence (perhaps several thousand bases). The user sees the sequence (maybe just the first/last parts for sanity) and clicks generate. The program then converts that sequence to the audio pattern (using base frequencies and separators) and then modulates into the chosen music. The result is a song subtly encoded with the BRCA1 gene frequencies. This showcases how the module brings real science data into the creative audio domain.

- **Error Handling:** Provide feedback if a search fails (e.g., gene name not found) or if a sequence is extremely long (some whole chromosomes are millions of bases – you might warn that processing could be slow or suggest using layering). Possibly implement a length limit or automatically chunk extremely long sequences into layers of manageable size (linking with the multi-layer logic above).
- **Extendability:** This module could be extended in the future to allow BLAST searches or alignments, but for now the focus is retrieval and conversion. The infrastructure (using Bio.Entrez and parsing sequences) sets the stage for any such enhancements.

Batch Processing and Data Organization

The program supports batch conversion of multiple files and treats directories as special organized inputs. This feature is designed to mirror genomic organization (e.g., multiple files can be multiple chromosomes) and to simplify handling large collections of data. It also introduces a logical file-to-audio management scheme where output audio files or layers are named and grouped correspondingly.

- **Multiple File Input:** Enable the user to drag-and-drop or select multiple files at once for processing. The program will enumerate through each file and perform the conversion to base frequencies. By default, multiple files could be processed sequentially: for example, if three files are added and no special layering is chosen, the program might generate three separate audio outputs (each with its own modulation into the chosen music or into separate music tracks). However, the more powerful feature

is combining them.

- **Directory as Chromosome:** If a directory is dropped or chosen, interpret it as a higher-level grouping (a “chromosome”). All files inside that directory will be considered parts of one larger unit. By default, treat each file in the directory as a segment to be concatenated in order (the simplest approach). For instance, if a directory contains files part1.bin, part2.bin, part3.bin, the program can concatenate their converted base sequences one after another to form one large sequence before audio generation (thus the directory becomes one continuous chromosome track). Alternatively, allow an option to layer them instead of concatenation (if layering makes more sense for the use-case). The user interface can indicate “Combine files in directory sequentially or concurrently?”
- **Automatic Organization for Large Data:** In cases where an input file is extremely large (e.g., gigabytes) or a directory contains many files, automatically organize the data into sub-parts. For example, split a huge file into chunks of 10,000 bases each and assign each chunk as if it were a separate file in a directory, then proceed as above. This is analogous to splitting a long DNA molecule into fragments (like artificial chromosomes). The program can label these chunks (Chromosome1_Part1, Part2, etc.) internally. This segmentation helps in not overloading the audio generation (processing in smaller pieces) and sets up the multi-layer or sequential processing as needed.
- **Batch Processing Workflow:** Provide a clear UI workflow for batch jobs. Possibly list the files and directories added in a table. For each entry, allow the user to select whether it should output to: (a) a separate audio track, (b) merged sequentially into the previous item’s track, or (c)

layered in parallel with others. This flexibility covers different use cases. A “Process All” button would then loop through the list and apply conversions. If separate outputs are chosen, each file could produce its own modulated music file (maybe the user wants to create a separate song per file). If merged sequentially, the program will join their base sequences and produce one song. If layered, the program will apply the multi-layer modulation technique to embed them together in one song.

- **Harmonic Layering of Multiple Files:** By default, if a directory is specified (multiple files as one chromosome), use **harmonic layering** for those files: instead of playing them one after the other, consider playing them simultaneously as layers, but tuned harmonically. For example, if there are four files in the directory, assign each an interval in a chord (one could be at base carrier 528 Hz, another maybe offset to create a perfect fifth above it, etc., or different subcarrier frequencies as earlier described). Then embed all in one music piece via nested modulation or mixing. This would truly produce a layered harmonic structure representing the entire directory’s data at once. This approach should be optional or used when “harmonic mode” is selected, because it could sound more dissonant if not carefully tuned. Ensuring that each layer’s frequencies complement each other (like choosing different octaves or frequency bands per layer) will achieve a more harmonious result, hence the term *harmonically layered*. Use simple musical intervals for offset if doing this (e.g., layer2’s frequencies could all be transposed a fifth higher than layer1’s, which in frequency is a $\times 1.5$ ratio approximately; layer3 maybe a major third above layer1, etc.). By doing so, when they play together, many of their tones will form consonant chords.
- **Output File Structure:** When exporting, maintain an organized naming

scheme. For example, if processing multiple files to separate outputs, name the result audio files after the input (e.g., song_myfile1.wav, song_myfile2.wav, or append the project name). If combining into one track, consider embedding metadata: e.g., in the output audio's metadata (ID3 tags or WAV metadata), list the names of files encoded. This is useful documentation especially for later retrieval or debugging (the user knows which files' data were included).

- **Large Dataset Example:** Suppose the user has a folder “GenomeData” with 23 files (perhaps representing chromosomes 1–22 and X of a genome). The program could allow selecting the entire folder, and then automatically treat each file as a chromosome layer. It could either generate 23 separate audio tracks (one per file) or embed some/all together. If the user wants one *massive* piece, the program might combine them in layers (maybe grouping to not have 23 simultaneous layers, which could be too complex). Perhaps it would sequentially arrange some and layer subsets. The design can be adjusted based on performance (maybe limit to layering 4 at a time in a chord, then move to next 4). The key is that the user doesn't have to manually handle each file – the software intelligently organizes and processes them.
- **Biologically-Inspired Compression:** This ties with large data handling. Implement an algorithm to detect repetitive elements across files or within a file. For example, if file1 and file2 share a large portion of identical bytes (or if within file1 a certain sequence of bytes repeats multiple times), note this. Instead of generating the same audio pattern redundantly, you could generate it once and then reference it. A simple approach: if two identical base sequences occur, generate one and for the second occurrence either skip generation (silence) or insert a short

placeholder tone indicating “repeat”. A more advanced approach could be to assign a layer specifically for repeats: e.g., layer1 plays a motif, layer2’s amplitude modulation instructs how many times to consider it repeated. This concept is quite advanced and may not be fully realizable in audio without losing information, but even partial implementation (like detecting and alerting “50% of this data is repetitive; consider using compression”) is useful. At minimum, ensure that obvious large runs (like a file full of zeros) don’t produce endless droning – one could collapse a run of identical bases into a single sustained tone plus an indicator.

- **User Feedback:** After batch processing, show a summary: “Processed 5 files into 2 audio outputs. Output1 contains File A, B (layered); Output2 contains File C (solo track)...” etc. This helps users verify the batch behavior.
- **Parallel Processing:** Use threads or async processing to handle multiple files concurrently if they are separate outputs, which can speed up batch jobs on multi-core CPUs. For combined outputs, much of the work is still per file (conversion) which can be parallelized before the modulation combination step.
- **Flexibility:** The UI might allow the user to toggle between treating multiple inputs as sequential vs parallel. For example, if someone drags 5 files, you could have checkboxes “stitch sequentially” or “layer in parallel.” If sequential is chosen, the program will combine their sequences end-to-end (maybe inserting a larger separator or a distinct tone between file sequences to mark boundaries, such as a brief silence or a specific frequency marker). If parallel is chosen, treat them as layers (with the harmonic modulation approach above).

- **Memory Management:** Generating audio for many files can be memory heavy. It might be prudent to stream results to disk instead of holding everything. For instance, if processing sequentially, after finishing one file's audio, you could write it to a temp file and then start the next, rather than keeping both in RAM. Use on-disk buffering or incremental writes for long sequences (this can be facilitated by libraries or manually writing WAV chunks). For layering, since you may need all layers aligned, you might have to hold multiple arrays; be mindful and perhaps limit how many layers simultaneously to not exceed memory.

Music Integration and Global Language Support

This section highlights features that enhance the program's versatility in working with diverse musical inputs and language outputs. It covers real-time musical analysis and pitch-matching, as well as integrating an offline translator to support many languages for text encoding.

- **Real-Time Music Pitch Detection:** The program should analyze the provided music track on the fly to guide modulation. Using a real-time pitch detection algorithm, it can track the chords or dominant notes as the music progresses. Techniques include using the **Librosa** library to perform short-time Fourier transform (STFT) or autocorrelation to detect the fundamental frequency over time. Another approach is using a **pitch tracking algorithm** (such as Yin or Mel-frequency cepstrum methods) to get the melody line. For our purposes, a simpler method is to identify the chord progression or at least the root note of each segment (e.g., each bar or beat). This can be done by segmenting the music by beats

(Librosa's beat tracking can find tempo and downbeats) and then for each segment computing the predominant frequencies. With this info, the program can adjust the genomic audio in real time: e.g., if during one bar the music is around an F major chord, the DNA frequencies that play during that bar could be nudged towards the nearest components of that chord (F, A, C frequencies). This **dynamic note matching** ensures the embedded frequencies continuously complement the music's harmony.

- **Adaptive Modulation Control:** The program can also use real-time analysis to modulate the intensity of embedding. For example, during loud or busy sections of the music, the DNA signal might be more deeply embedded (slightly higher amplitude) because it will be masked by the music, whereas in quiet sections, reduce the modulation depth further to keep it hidden. The volume or spectral density analysis of the music can drive this. Essentially, tie the modulation depth to the inverse of the music's loudness at any given moment.
- **Pitch Tuning Reference:** The music analysis should determine the tuning reference as mentioned (440 vs 432 or other). This can be done by finding the deviation of detected pitches from nearest equal tempered grid. Librosa has `librosa.estimate_tuning()` which returns the deviation in fractions of a semitone for the music. Use that to decide if retuning is needed. If the user has enabled "Auto-retune music to A=432", apply a high-quality resample/pitch shift to every frame of the music by the negative of that deviation plus any additional offset to reach 432. Keep the option to disable this for purists who want the original tuning preserved.
- **Global Language Translation Support:** Integrate a translation library so that if a user inputs text in any language, it can be converted to the

Hebrew (or other target) for encoding, or even directly to other languages if future mapping schemes are added. **Argos Translate** is a good choice as it is open-source, offline, and supports dozens of languages with no required attribution . Include the Argos Translate models for at least the major languages to Hebrew and Hebrew to English (in case back-translation is needed for verification). The workflow would be: detect the language of input text (Argos may have detection, or use a simple heuristic or a separate library). If it's not already Hebrew and the user has enabled translation, call Argos to translate from the detected language to Hebrew. If translation to Hebrew is not available for that language, perhaps translate first to English, then English to Hebrew (Argos can chain translations through English as an intermediate). This way, a user could input, say, Spanish text, and it would go Spanish -> English -> Hebrew internally, then to DNA codons. All done offline.

- **Extending to Other Alphabets:** While Hebrew is a requirement here, the translator allows you to support any language. If needed, you could also support alternate encoding schemes: e.g., maybe implement a similar 3-base encoding for Cyrillic or Greek if those alphabets were of interest, using Argos to get the content into those scripts. The UI might have an option for “Encoding alphabet” with currently “Hebrew (DNA/Gematria)” and potentially future ones.
- **No Attribution Constraint:** Since the user requested no attribution requirements, ensure the chosen translator and any language libraries are permissively licensed (Argos is MIT). Do not use online APIs like Google Translate (which would violate offline requirement and licensing). Argos Translate being local satisfies privacy and licensing concerns.
- **Use Case (Language):** A user could input a poem in French, select

“Encode linguistically,” and the program will translate the poem to Hebrew, then map to DNA bases and generate the frequencies. The output song thus carries the “essence” of the poem encoded in Hebrew phonetic frequencies. This is a unique feature combining language and sound.

- **UI Implementation for Language Support:** Provide dropdowns or detection for input language and a toggle for translation on/off. Perhaps also allow the user to choose the target language for encoding (default Hebrew). For example, one could even choose to encode in another alphabet if supported (though by default, the spec is focusing on Hebrew).
- **Testing and Quality:** The translator’s output can be verified by back-translating a portion to ensure it makes sense (not necessary for the program’s functionality but for user trust). Maybe show the translated Hebrew text to the user in the interface (some users might read Hebrew and can confirm the translation captures the meaning).
- **Performance:** Argos Translate is reasonably fast for short texts but ensure that if a very large text is given (like a whole book), the translation might take time. Use a separate worker thread for translation to keep the UI responsive and show a progress bar (“Translating text...”). Once done, proceed to mapping and audio generation.
- **Dynamic vs Static Embedding of Language:** Since the audio is ultimately frequency data, the meaning is “encrypted” in it via the codon mapping. If someone knew the mapping, they could decode the Hebrew message from the audio by analyzing the frequencies. The program could optionally output a legend of which frequency corresponds to which

letter (since each codon frequency triplet corresponds to a Hebrew letter via gematria mapping). However, given the complexity and the fact that multiple frequencies per letter might exist, we might not expose that level of detail unless needed.

- **Speech or Phoneme Integration (Possible Extension):** As a creative extension, one might consider mapping not just letters but actual phonetic sounds to frequencies, enabling a form of hidden speech in the audio. This is beyond the scope of the current plan but worth noting as a future idea: e.g., use an algorithmic text-to-speech to get phonemes and assign each phoneme a frequency pattern. But for now, the letter-based encoding is sufficient and aligns with the Gematria concept.

Output Formats and Audio Export

The program must support exporting the final modulated audio in various high-quality formats to accommodate different user needs (archival, further editing, or sharing). Key points for implementation include providing format choices, ensuring maximum audio fidelity (bit depth, sample rate), and using appropriate libraries for encoding without violating licensing.

- **Supported Formats:** Provide export options for **WAV, FLAC, AIFF, MP3,** and **AAC**. These cover common uncompressed, lossless compressed, and lossy compressed formats. In the UI, this can be a dropdown menu or a set of radio buttons for the user to choose the format after generation.
- **WAV (PCM) Export:** WAV should support up to 32-bit depth and 192 kHz sampling rate as specified. Typically, the audio is being processed

internally at high resolution (we aimed for 192 kHz, 32-bit float). When exporting to WAV, you can quantize to 32-bit integer PCM or 32-bit float PCM if supported (32-bit float WAV is also common). Use Python's built-in wave module or the **soundfile** (pysoundfile) library to write WAV files with the desired format settings. Ensure the format chunk is correctly set for 32-bit PCM. Provide options in the UI for bit depth (e.g. 16, 24, 32) and sample rate (44.1, 48, 96, 192 kHz) in case the user wants to downsample or use a specific setting. By default, use 32-bit 192 kHz for maximum quality (this will produce large files, but the user specifically mentioned up to that quality).

- **FLAC Export:** FLAC is a lossless compressed format that will reduce file size without losing quality. Use a library like **pysoundfile** or an external encoder. Many audio libraries can write FLAC if libFLAC is installed. If using soundfile, simply give a .flac extension and it will compress. Support up to 24-bit in FLAC (FLAC supports up to 24-bit audio typically, and sample rates up to 192k). If the internal audio is 32-bit float, you might need to convert to 24-bit int for FLAC (since 32-bit int is not well supported by FLAC encoders). That conversion can be done by scaling and rounding. The UI should perhaps restrict bit depth to 24-bit when FLAC is chosen.
- **AIFF Export:** AIFF is an uncompressed format similar to WAV, often used on macOS. Python's soundfile library can write AIFF, or use an AIFF writer. Support similar bit depths (up to 32-bit) and sample rates. AIFF-C (the extended AIFF) can handle 32-bit float if needed. Ensure testing on Mac that the output is readable by standard audio software.
- **MP3 Export:** MP3 is a lossy format. Use **pydub** or **ffmpeg/Libav** via subprocess to encode to MP3, since Python doesn't have a native MP3

encoder due to patent issues (though by 2025 patents are expired, but library support might still rely on LAME). The easiest integration is to require the user have ffmpeg installed or bundle an ffmpeg binary, and call it to do the encoding. Alternatively, use **pydub** which can export to MP3 if ffmpeg is configured. Offer bitrate options (e.g. 320 kbps, 256 kbps, etc. via a dropdown). Since the user mentioned Creative Commons licensing for the project, including an open-source encoder like LAME (LGPL) should be fine as long as licensing is noted. No attribution needed but we should still comply with LGPL by allowing dynamic linking (if we bundle a DLL or rely on external ffmpeg).

- **AAC Export:** AAC (in M4A container typically) is another lossy format, widely used (especially for Apple devices). Similarly, use ffmpeg or an AAC encoder to create this. AAC has profiles (just default to AAC LC). Provide options for bitrate or quality. Using ffmpeg command like `ffmpeg -i input.wav -c:a aac -b:a 320k output.m4a` can produce an AAC. Or use Apple's CoreAudio on macOS if accessible (though going that route complicates cross-platform support – better to stick with ffmpeg).
- **UI for Export Settings:** In the interface, after generation, show an export panel where the user can select format and quality settings before saving. Or have these settings pre-selected and then just produce that format. Possibly provide a checkbox “export stems” or “export each layer separately” in case user wants individual layer audio (for advanced users who might remix it – this could output, for example, the pure DNA tone track, the FM carrier track, etc., separately). However, that's an advanced feature; by default, just final mix export.
- **Ensuring Precision in Export:** When converting to fixed bit depth (like 16-bit), use dither to avoid quantization distortion, especially since our

audio has very low-level signals embedded. A TPDF dither can be applied when going to 16-bit. For 24-bit or 32-bit, quantization error is negligible for our needs, but it's still good practice. The high sample rate (192k) might not be necessary for the final output if the music was originally, say, 44.1k – but since we might have introduced supersonic frequencies by speeding up DNA or by having many layers, exporting at 192k ensures we keep everything. The user can down-sample later if needed. We should, however, provide the ability to export at the original music rate too (for a transparent embedding that matches original track exactly in length and rate).

- **Tagging and Metadata:** If possible, write metadata tags indicating this file was generated by “36N9 Genetics Audio Genomics Program” and perhaps listing the project or encoded content. For example, set the ID3 Artist to “36N9 Genetics LLC” and Comment to “Contains encoded DNA audio.” The branding requirement suggests we should put the company name somewhere visible – metadata is a good, non-intrusive place. (We will also display it in the UI as per UI section).
- **Testing Compatibility:** Test that the exported files play in common media players. Particularly, 32-bit 192k WAV and AIFF should be tested (some older players might down-sample or not play 192k or >24-bit, but professional tools will). FLAC and MP3, AAC should be widely compatible at standard bitrates.
- **Performance of Encoding:** Encoding to MP3/AAC will take some time for long files. Possibly do this in a background thread as well, with a progress bar (especially if using ffmpeg, you can read its stdout for progress or just approximate based on file length). WAV/AIFF exports are relatively quick (just writing to disk).

- **Library choices:** For WAV/AIFF/FLAC, the Python **soundfile** library (which uses libsndfile) is ideal as it handles all of those with one API and is BSD licensed. For MP3/AAC, rely on **ffmpeg/avlib**. Make sure to include instructions for the user to install ffmpeg if not bundled (or bundle a static ffmpeg binary for convenience).
- **Multiple Simultaneous Exports:** Possibly allow exporting in multiple formats at once (e.g., user could tick WAV and MP3 and get both). This is a nice-to-have; otherwise they can export one then choose another and export again.
- **File Naming:** Default the output file name to something based on the project name or music file name plus an indication of encoding. For example, if the music is “song.wav” and we embed data from “file1”, the output could default to “song_genomic_mix.wav”. For batch mode where multiple outputs are produced, incorporate the input identifiers in the name.

Graphical User Interface Design

The user interface should be intuitive, futuristic, and professional – resembling a lab-grade instrument software. It must accommodate all the tools (file input, conversion settings, modulation settings, gene search, etc.) in an organized manner using modern GUI components like tabs and panels. The design will prominently feature the brand **36N9 Genetics LLC** and incorporate advanced UI elements such as drag-and-drop and dynamic unfolding panels.

- **Overall Theme and Style:** Use a **futuristic** aesthetic – likely a dark background with high-contrast text and neon or metallic accent colors (this gives a “lab equipment” or sci-fi feel). Consider a styled Qt widget theme or CSS for a modern look. The layout should look professional (no crowded controls, logical grouping) as if it’s software used by genetic audio researchers.
- **Main Window Layout:** The main window could be divided into sections or tabs:
 - A welcome/home tab with the branding and perhaps a brief description or logo.
 - A **“Data Input” tab** where users add files or text and choose conversion options.
 - An **“Analysis/Conversion” tab** showing the Hebrew/text conversion or gene fetch results (if applicable).
 - A **“Modulation & Music” tab** where the user loads a music file, and sets modulation parameters (like volume mix, retune option, etc.).
 - A **“Output/Export” tab** for choosing format and exporting.
 - Alternatively, a single-page interface with collapsible sections (like accordions) titled “1. Input Data”, “2. Genomic Encoding”, “3. Modulation Settings”, “4. Output Options”.
- **Drag-and-Drop:** Implement drag-and-drop for files and directories onto the relevant area (e.g., a drop zone in the Input tab). When a user drags a file in, highlight the drop area (“Drop files to encode”). On drop, automatically list the files and possibly parse them (e.g., if a text file is

dropped, read a snippet to confirm it's text). For directories, do similarly. This makes adding input extremely easy.

- **Tabs and Scrollable Panels:** Where content might overflow (like a long list of files or a long sequence text), use scrollable areas. For example, if a user fetched a gene sequence, the sequence text area should scroll. Use tabs to separate major functions (like “Sequence Editor” vs “Settings”).
- **Origami-Style Unfolding Panels:** This refers to UI panels that expand or unfold when activated, perhaps with an animation. For example, the program could initially show only basic settings, and have an “Advanced settings” button that unfolds (animates open) more options (like a panel sliding down). Use this for rarely changed parameters (like fine-tuning modulation depth, choosing specific codon mapping scheme, etc.) so the main interface stays clean. Qt supports expand/collapse panels (e.g., QSplitter or animation on size).
- **Pop-up Tools with Progress Bars:** Long-running tasks (translation, audio generation, modulation processing, file encoding) should trigger a small pop-up or overlay dialog showing progress. For example, when the user clicks “Generate Audio”, open a progress dialog saying “Generating audio... 45%” with a cancel button. Similarly, when downloading from NCBI or translating text. These pop-ups can have a sleek design (semi-transparent window overlay maybe). Use Qt’s QProgressDialog or a custom dialog for this. Ensure that if the user cancels, you stop the process thread gracefully.
- **Branding Placement:** Display “**36N9 Genetics LLC**” prominently. This could be in the title bar of the application (e.g., “36N9 Audio Genomics

Suite”), and also within the GUI, perhaps as a logo at the top or bottom. Possibly include a small logo image if provided (if not, a stylized text label). On the home tab or about dialog, put “© 2025 36N9 Genetics LLC” and any tagline. This ensures the brand is always visible during use. If possible, also embed the name into the audio metadata as mentioned in Export.

- **Platform Adaptation (macOS):** Since target is macOS (both Intel and Apple Silicon), ensure the UI uses standard macOS conventions: e.g., have a menu bar with at least an “About” and “Quit” item, make the main window properly sized for Mac screens, use Mac-style file dialogs (Qt will handle that). For Apple Silicon compatibility, ensure that any bundled binaries (like ffmpeg or other libs) have universal builds or ship separate versions. Qt for Python (PySide6) supports Apple Silicon natively; we just need to package correctly. The GUI design itself should be resolution-independent (use Qt’s layouts so it scales on Retina displays nicely).
- **Responsiveness:** The GUI should remain responsive during processing. Achieve this by performing heavy tasks on background threads (using Python threads or Qt’s QThread, or multiprocessing for CPU-bound tasks). The progress dialogs will be updated via signals/slots. This way, the user can still interact (maybe to some extent) or at least the UI doesn’t freeze. For example, one could queue multiple tasks: the user could start generation for one dataset and while it’s processing, perhaps prepare another dataset on a different tab.
- **Feedback and Logs:** Have a text area or console output section (perhaps hidden behind a “Log” toggle) to display messages, such as “Loaded file X, 1024 bytes -> 4096 bases”, “Translated text to Hebrew:

אבא...", "Fetching gene NM_XXXXX from NCBI..." etc. This helps advanced users see what's happening under the hood. It's also useful for debugging.

- **Error Handling UI:** If something goes wrong (file format not recognized, ffmpeg missing, etc.), show an error dialog with a clear message. For instance, "Error: ffmpeg not found for MP3 export. Please install ffmpeg or choose a different format." This prevents silent failures.
- **GUI Elements Summary:**
 - **Menu Bar:** with options like File (Open, Save Project, Exit), Tools (maybe some toggles), Help (About, Documentation link).
 - **Toolbar:** could have quick buttons for "Add File", "Add Directory", "Fetch Gene", etc., represented with icons.
 - **File List Panel:** listing added files/dirs with icons indicating type (text vs binary vs directory), and maybe a column for how it will be handled (like "Binary->DNA" or "Text->Hebrew->DNA" etc.). Perhaps allow selecting a list item to bring up its details (like preview text or stats).
 - **Settings Panel:** where user chooses things like "DNA vs RNA output", "Use Hebrew encoding for text [yes/no]", "Retune music to 432 Hz [yes/no]", modulation depth slider, etc.
 - **Gene Fetch Panel:** with a text field for search term and a "Fetch" button, plus options for sequence type (nucleotide vs protein).
 - **Sequence Text Editor:** as described, for manual editing or viewing fetched sequences.
 - **Music Panel:** to load the music file and display basic info (waveform

or at least duration, maybe a small spectrum). Could also have a play button to preview the music or the final mix within the app (if we integrate an audio playback feature using something like QtMultimedia).

- **Progress/Status Bar:** at the bottom of main window, show brief status (“Ready.”, “Converting file1.bin...”, etc.), and maybe a small progress bar when tasks run (if not using pop-ups).
- **Mac Packaging:** Finally, for macOS distribution, package the app as a .app bundle including all dependencies (PyInstaller or cx_Freeze can help create a standalone app). Test on both Intel and M1 Macs. Since PySide6 supports universal builds, we might have to build on each architecture or lipo combine binaries. If using Tauri (an alternative to Qt, with a web UI), the approach would differ, but given the complexity of DSP here, Qt/Python is a straightforward route.

Licensing and Legal Considerations

It is crucial to adhere to the licensing terms for both the project’s output and all incorporated libraries/tools. The project will be released under a Creative Commons license with specific requirements, and we must ensure all components are compatible with that. Additionally, the program should inform users of any commercial use obligations.

- **Project License (CC BY 2.0):** The entire codebase and deliverables of this program are to be released under the Creative Commons Attribution 2.0 License, as specified by 36N9 Genetics LLC. This means anyone can share or adapt the software as long as they credit 36N9 Genetics LLC as

the original creator. Include a LICENSE.txt file in the project root stating this license. Clearly mention “© 2025 36N9 Genetics LLC – Released under CC BY 2.0” in the documentation and About dialog. Even though CC BY is not typically used for software (usually MIT/GPL are), we will abide by the request.

- **Royalty Clause for Commercial Use:** In addition to CC BY, the user stipulates a 20% revenue royalty for commercial use. This is not a standard CC BY term, but rather an extra condition. To implement this, include a section in the license or a separate “Commercial Use Addendum” that says: if this software (or any derivative) is used commercially (sold or used in revenue-generating products/services), 20% of the revenue must be paid to 36N9 Genetics LLC. This effectively makes the license a custom CC BY + royalty license. Make sure this condition is prominently displayed to users (for instance, in the About dialog or first-run notice). It might also be wise to include a click-through agreement on first launch for users to acknowledge this if they plan commercial use.
- **Library Licenses:** We have chosen libraries that are permissive (public domain or MIT/BSD style) to avoid attribution requirements that conflict with a no-attribution goal. For each third-party library used, double-check its license:
 - **PySide6 (Qt for Python):** LGPL 3.0 – this is allowed as long as we dynamically link (which we do by using it normally) and provide a way to relink (basically, LGPL is okay for use in proprietary or CC software, no need for attribution in product, but we should mention using Qt somewhere). No royalty issues.

- **Librosa (ISC license), NumPy/SciPy (BSD), pydub (MIT), Biopython (Biopython License, similar to BSD), Argos Translate (MIT)** – all these are permissive. We should include copies of their licenses in a `THIRD_PARTY.md` or in the docs to be safe, even if attribution isn't legally required in documentation, it's good practice. The user requested no attribution for libraries, which we interpret as the user doesn't want to have to attribute others in their usage of the program. These licenses generally only require including the license text in distribution, which we can do without burdening the end-user.
- **Liquid-DSP (MIT)** if used, similarly include license file.
- **FFmpeg/LAME** if bundled: FFmpeg is LGPL 2.1 (or GPL depending on compile). We can include an LGPL build of ffmpeg so it's legally safe to include without source (just need to offer to provide source). LAME is LGPL. We will include their license texts and an offer in documentation to get the source code for those binaries, fulfilling LGPL requirements.
- Ensure none of these licenses conflict with the CC BY + royalty. Generally, including LGPL or MIT code inside a CC BY project is fine as long as we abide by their terms. CC BY is more restrictive (attribution) than MIT for example, but since we are attributing ourselves mainly, it's okay. The royalty clause however does not propagate to those libraries (we cannot impose royalty on someone else's MIT code usage, but since they get it only through our app, it's fine).
- **Attribution in Documentation:** Although user desires no attribution overhead, we should still credit sources in an "About" section quietly. For

instance: “This software uses open-source components: Qt, NumPy, etc. See THIRD_PARTY.md for details.” This is typically acceptable and does not burden the user, and it respects the licenses.

- **Copyright Notices:** Mark all source files with a header: e.g. “Copyright 2025 36N9 Genetics LLC. Licensed under CC BY 2.0. For third-party licenses, see THIRD_PARTY.md.” Also include the CC BY logo or link to human-readable license text in documentation.
- **Commercial Use Enforcement:** Practically, the program can’t enforce the 20% royalty itself (that would be a legal matter outside the software). However, to remind users, the About dialog or license file should explicitly state: “Commercial use of this software or derivatives requires a 20% royalty to 36N9 Genetics LLC.” Perhaps provide contact info for licensing questions. This will put users on notice, though enforcement would rely on legal agreements rather than code.
- **Disclaimer:** It’s wise to include a disclaimer of warranty (since CC BY doesn’t explicitly include one as GPL does). Add a section: “This software is provided as-is, without warranty of any kind. 36N9 Genetics LLC is not liable for any outcomes from using this software.” This is standard to protect the company.
- **User Privacy:** Note if any data is sent out (the only case would be NCBI fetch which obviously sends query to NCBI). In documentation, mention that gene fetching will contact NCBI’s servers and subject to their privacy policy. Otherwise, everything is offline/local.
- **Final Step - Packaging License Info:** When building the .app or installer, include the license agreement text. For Mac, you might present the license in the installer package step or just bundle the license files

with the app. As part of the installation or first-run, have the user agree to the CC BY + royalty terms (especially to highlight the commercial royalty part).

- **36N9 Trademark:** The branding usage in software should be fine (the company presumably gave us rights to use their name/logo). If we include any third-party trademark (none foreseen, except maybe “NCBI” mention or “Argos”), ensure it’s just nominative use.
- **Continuous Compliance:** As development proceeds, review any new library we consider: only include it if it’s permissive. If something absolutely required had a restrictive license (GPL), we’d either avoid it or separate it so as not to affect the whole program’s license. Fortunately, our choices avoid that.
- **User documentation:** Provide a user manual or README that also reiterates license terms and how to credit if they share results (e.g., if they share an audio file created, do they need to credit? Probably not, since the CC BY is on the software, not necessarily on the output audio, unless the output contains creative content from the program’s authors – arguably the mapping scheme and audio output might be considered a creative work too. Clarify whether the *generated audio* is also under CC BY or if it’s the user’s own. Possibly clarify: the software is CC BY, but any audio you create with it is your own (unless the DNA sequences have copyright – but genetic sequences aren’t copyrightable, and audio frequencies aren’t either really). So likely the output belongs to the user. This distinction can be mentioned to avoid confusion.)

Implementation Strategy and Dependencies

To achieve all the above functionality efficiently, the program will leverage a number of open-source libraries and tools, stitched together in a Python-based architecture for quick development. The implementation approach is summarized here, mapping features to chosen technologies and ensuring performance:

- **Programming Language:** Python 3.x will be the core language, given the availability of scientific libraries (NumPy, SciPy), audio processing libraries (pydub, librosa), and GUI frameworks (Qt for Python). Python's flexibility and rapid development fit the requirement, and it runs on macOS (both Intel and Apple Silicon) without issue. For performance-critical DSP (like generating long waveforms or doing real-time modulation), we will use vectorized NumPy operations in C speed, or offload to C libraries.
- **Audio Processing Libraries:**
 - Use **NumPy** for signal generation (creating wave arrays, doing modulation math) because it's fast and precise.
 - **SciPy** or **Matplotlib** can be used if any signal filtering or plotting is needed (maybe for an optional waveform display).
 - **pydub** (MIT license) will be used for easy audio format handling (loading MP3s via ffmpeg, slicing, overlaying tracks). Pydub's high-level interface simplifies combining the music and modulator signals and exporting through ffmpeg.
 - **Librosa** (ISC license) for music analysis tasks like pitch detection, beat tracking, and tuning estimation. Librosa builds on NumPy and will

help implement the auto key detection and retuning functionality .

- For **tone synthesis**, instead of coding every waveform by hand, we might use **SoX** (Sound eXchange) as a backend by calling it for complex waveforms. However, since SoX is GPL and also an external binary, we prefer generating waveforms ourselves with NumPy (e.g., $\text{sine} = \sin(2\pi ft)$, $\text{square} = \text{sign}(\sin())$, $\text{triangle} = \text{asin}(\sin())$ etc.). This avoids licensing issues and keeps everything in-process. If high precision is needed for frequency, Python's `decimal.Decimal` can be used to compute the frequency value, then cast to float for actual sample generation. The difference at 38 decimal places is negligible in terms of sample values at 192kHz, but we keep the spirit by not rounding frequencies prematurely.
- **Liquid-DSP** (MIT license) for FM modulation: We can include this C library and call its functions via Python's `ctypes` or `ctypes`, or possibly use a Python wrapper if one exists. Liquid-DSP provides ready-made FM modulators which could simplify the FM step. If integrating proves complex, we will implement FM by manipulating phase as described using NumPy – which is feasible given modern CPU speeds, even for lengthy signals.
- **Data Handling:** Use Python's built-in libraries for file I/O (reading binary, text). For example, reading a binary file and converting to bits can be done with Python bit operations or using NumPy `frombuffer`. For translation, the Argos library will handle model loading and inference.
- **GUI Framework: Qt for Python (PySide6)** is chosen for the GUI. It's robust, cross-platform, and can achieve the required interface (drag-drop, tabs, custom styling). PySide6 is LGPL, which is fine for our distribution.

We will design the UI using Qt Designer for speed or code it by hand with QtWidgets. Either way, PySide allows styling via Qt Style Sheets for the futuristic look.

- **Threading Model:** Use Python threads (with care to avoid GIL issues – but heavy NumPy computations release the GIL) or Qt's QThread for offloading tasks. We might use Python's concurrent.futures ThreadPool for simplicity to run tasks like “generate audio” or “encode mp3” in parallel. We will synchronize with the GUI thread using Qt signals or a callback mechanism to update progress.
- **External Tools Integration: FFmpeg** will be included for encoding MP3/AAC (unless we restrict to calling an external installation). To avoid complicating user setup, we can bundle ffmpeg. Since ffmpeg is large, we might provide an instruction for the user to install it themselves if they choose MP3/AAC; but a better UX is bundling a thin build of ffmpeg (for Mac, could use static builds). We must abide by LGPL by offering source if requested.
- **Precision and Performance Testing:** We will test frequency generation by generating a known sequence and verifying frequencies with an FFT to ensure they are exact (e.g., generate Adenine tone and verify a peak at 545.6 Hz). Use double precision floats in NumPy (default) to maintain high precision in calculations. 38 decimal places precision is far beyond float64 (~15 decimal), but the actual requirement likely is just to avoid noticeable detune. We'll ensure that the frequencies are accurate to at least 0.000000001 Hz which is certainly met.
- **Memory management:** For potentially hours-long audio, generating a full NumPy array could be many million samples (e.g., 5 minutes at

192kHz is ~57 million samples, which is fine; but an hour is ~691 million, which is ~5.3GB at 32-bit float!). To be safe, for extremely long sequences or multi-layer combos, we might generate and process in chunks (streaming). Python generators or writing to disk in chunks (via soundfile) could be used to not hold entire huge arrays at once. But for moderate lengths and modern RAM sizes, we can usually hold the final wave in memory if under a few GB. Nonetheless, caution: if someone tries an entire chromosome (~50 million bases) at 3 cycles per base at 192kHz, that's enormous. In such cases, we rely on our earlier logic to compress or split layers. That way we never actually try to synthesize billions of samples at once.

- **Testing on Mac (Intel & M1):** We will test the app on both architectures. Using PySide6 and pure Python means we can run on Apple Silicon natively (assuming we install an ARM-native Python and libs). For distribution, we might create two app bundles (one built on Intel with universal binaries, or one on each arch). It might be easier to build on Intel Mac and use PyInstaller's `--target-arch arm64` if supported, or vice versa. Alternatively, instruct M1 users to use Rosetta if needed, but ideally deliver universal. We will try to include universal binaries for anything compiled (ffmpeg can be fat-lipo combined).
- **Development Plan:** Implement features incrementally:
 1. Core conversion and tone generation (test with a small DNA sequence and output as WAV).
 2. Add modulation steps (FM, then AM) and test embedding into a simple carrier (maybe a sine wave as music).
 3. Integrate a sample music file and test the full pipeline, listening to ensure

the music isn't distorted audibly.

4. Implement UI gradually: first file loading and output, then add translation and gene fetch, etc.
 5. Optimize as needed after functionality is complete.
- **Documentation & Usage Instructions:** Prepare a user guide explaining how to use each feature, the meaning of each option, and noting the license terms. This can be a PDF or Markdown shipped with the app, and/or an online help link.
 - **Final QA:** Verify that all specified features are present and working: dynamic tones for bases, correct insertion of 528 Hz separators, binary/hex mapping correctness, Hebrew translation accuracy, modulation audible (in a controlled test) yet subaudible in music, layering combining data without obvious artifacts, UI doesn't crash, and output files are in requested format. Also ensure the branding and license info appear properly.

By following this comprehensive build plan step-by-step, an AI code builder (or development team) can systematically implement the advanced audio genomics program, resulting in a cutting-edge application that turns DNA/RNA sequences into harmonious audio, all within a user-friendly, well-structured interface. Each component above should be carefully coded, integrated, and tested, leveraging the chosen libraries for reliability and efficiency. The end result will be an innovative tool that embodies the fusion of genetics, music, and technology as envisioned.